Serial Mock Documentation

Release 0.0.3

Joran Beasley

Dec 05, 2017

Contents

I wrote SearchableCollections in order to provide an ORM like interface to regular lists

CHAPTER 1

Requirements

- Python2.6, Python2.7, or Python3

CHAPTER 2

Installation

`setup.py install`
or `pip install .`
or install it from pipy with `pip install searchable_collection`
or directly from github `pip install git+https://github.com/joranbeasley/searchable_collection.git`

CHAPTER 3

Examples

3.1 examples using python primatives

3.1.1 Creating A SearchableList

you can create a list just like a normal list (well mostly)

```
from searchable_collection import SearchableCollection
some_other_list = [1,2,3,4,5,6]
my_list = SearchableCollection(some_other_list)
print(list(my_list.find_all_where(in=[4,5])))
```

or you can simply append items as needed

```
from searchable_collection import SearchableCollection
some_other_list = [1,2,3,4,5,6]
my_list = SearchableCollection()
for i in some_other_list:
    my_list.append(i)
print(list(my_list.find_all_where(in=[2,6])))
```

or you can use extend

> from searchable_collection import SearchableCollection some_other_list = [1,2,3,4,5,6] my_list = SearchableCollection() my_list.extend(some_other_list) print(list(my_list.find_all_where(in=[2,6])))

3.1.2 What can go in a Searchable Collection?

well pretty much anything... and it should just work, originally it was designed specifically with classes in mind, however it should really work just fine with anything

```
original_data = [[1,2,3],[3,4,5,'e'],{"w":7},"pie","apple",{"e":67},1,2,3,4,5,6]
my_list = SearchableCollection(original_data)

print(list(my_list.find_all_where(e=67)))
print(list(my_list.find_all_where(contains="e")))
print(list(my_list.find_all_where(contains=2)))
print(list(my_list.find_all_where(contains=3)))

# do an re.match (only matches "pie")
print(list(my_list.find_all_where(match="p.e")))
# do an re.search (matches both "pie" and "apple")
print(list(my_list.find_all_where(search="p.e")))
```

it starts getting even more interesting with nested dictionaries

```
my_list = SearchableCollection()
my_list.append({"sub_dict":{"anumber":56,"aword":"apple","alist":[1,2,3]}})
my_list.append({"sub_dict":{"anumber":26,"aword":"pineapple","alist":[7,8,9]}})
my_list.append({"sub_dict":{"anumber":126,"aword":"orange","alist":[7,18,19]}})

# d['sub_dict']['anumber'] == 26
print(list(my_list.find_all_where(sub_dict__anumber=26)))

# d['sub_dict']['anumber'] > 50
print(list(my_list.find_all_where(sub_dict__anumber_gt=50)))

# d['sub_dict']['aword'] == "orange"
print(list(my_list.find_all_where(sub_dict__aword="orange")))

# "n" in d['sub_dict']['aword']
print(list(my_list.find_all_where(sub_dict__aword__contains="n")))

# d['sub_dict']['aword'].endswith("le")
print(list(my_list.find_all_where(sub_dict__aword__endswith="le")))

# 3 in d['sub_dict']['alist']
print(list(my_list.find_all_where(sub_dict__alist__contains=3)))
```

See also:

Query Reference

SearchableCollection API Documentation

3.1.3 What Modifiers Can I Use

the complete list of modifiers is as follows

```
__contains    -  x in y
__in          -  y in x # note that if the field is ommited it is replaced with is_in `...where(is_in=...)`
__startswith -  x.startswith(y)
__endswith    -  x.endswith(y)
__search      -  re.search(y,x)
__match       -  re.match(y,x)
# numeric operators
```

```
__gt         -  x > y
__gte        -  x >= y
__lt         -  x < y
__lte        -  x <= y
__eq         -  x == y # in general this is the assumed operation and can be ommited
```

you can optionally negate any of the operators

```
__not_contains   -  x not in y
__not_in         -  y not in x
__not_startswith -  not x.startswith(y)
__not_endswith   -  not x.endswith(y)
__not_search     -  not re.search(y,x)
__not_match      -  not_re.match(y,x)
# numeric operators
__not_gt         -  not x > y  # or x <= y
__not_gte        -  not x >= y # or x < y
__not_lt         -  not x < y  # or x >= y
__not_lte        -  x <= y     # or x > y
__not_eq         -  x != y
```

- genindex
- search

3.2 SearchableCollection Usage Guide

3.2.1 Simple Access

you may access a Searchablelist exactly the same as a normal list for the most part

```python
from searchable_collection import SearchableCollection
some_other_list = [1,2,3,4,5,6]
my_list = SearchableCollection(some_other_list)
print(my_list[2],my_list[-1])  # 3 and 6
print(len(my_list),my_list.pop(3),len(my_list))
my_list.append(5)
print(len(my_list),my_list[-1])
```

3.2.2 Adding Element To Searchable List

you should be able to add elements to a Searchable list the same as if it were a normal list

```python
from searchable_collection import SearchableCollection

my_list = SearchableCollection()
my_list.append(4)
my_list.extend(["a",66,{'asd':'dsa','b':5}])
```

3.2.3 Searching For Elements

this is really the whole purpose of this module, to provide a flexible ORM like interface to searching though lists I doubt its super efficient, so i wouldnt recommend using it with huge lists, but it should be able to search a few hundred records near instantly

See also:

Query Reference

Single Nested Element Search

When searching we can use all of our *Comparison Search Modifiers*.

```
from searchable_collection import SearchableCollection
raw_data = [1,2,3,"pie","apple",[1,2,"e",3],[3,4,5],{"x":7}]
my_list = SearchableCollection(raw_data)

# lets find all the items that have an e
items_with_e = my_list.find_all_where(contains=e)

# lets find all the items that are in [1,"pie",[3,4,5]]
items_in_list = my_list.find_all_where(is_in=[1,"pie",[3,4,5]])

# lets find all the items that startwith "a"
items_startwith_e = my_list.find_all_where(startswith="a"))

# we can also negate ANY of our modifiers
# lets find all the items that DO NOT startwith "a"
items_startwith_e = my_list.find_all_where(not_startswith="a"))

# lets find all the items that endwith "e"
items_endwith_e = my_list.find_all_where(endswith="e"))

# lets find all the items that are less than 3
items_lessthan = my_list.find_all_where(lt=3))
```

Single Nested Attribute Search

Like Single Nested Element Search we can still use all our *Comparison Search Modifiers*. but this time we will be accessing the attributes of a class

the format that we need to use for this is

```
find_all_where(<attribute_name>__<modifier> = <value>)
#the modifier is of coarse optional
find_all_where(<attribute_name> = <value>)
#or the modifier can be negated
find_all_where(<attribute_name>__not_<modifier> = <value>)
```

```
from searchable_collection import SearchableCollection
raw_data = [{"x":i,"y":j} for i,j in zip(range(25),range(100,74,-1))]
my_list = SearchableCollection(raw_data)

# lets find all the items that have x == 5
items_with_x5 = my_list.find_all_where(x=5)

# lets find all the items that have x <= 5
items_lte_5 = my_list.find_all_where(x__lte=5)

# lets find all the items that have x <= 5 && y > 97
```

```
items_lte_5 = my_list.find_all_where(x__lte=5,y__gt=97)

# lets find all the items that have x <= 5 && y != 97
items_lte_5 = my_list.find_all_where(x__lte=5,y__not_eq=97)
```

Multi Level Nested Attribute Search

now imagine we had some objects like the following, and of coarse you can still use all the *Comparison Search Modifiers*

```
class TestClass():
     def __init__(self,a,b,c,d):
         self.a=a
         self.b_list = {"b":b,"c":{"val":c,"next":d}}
     def __repr__(self):
         return str(self)
     def __str__(self):
         return "<TC="+str([self.a,[self.b_list['b'],[self.b_list['c']['val'],self.b_
↪list['c']['next']]]])+">"

objects = SearchableCollection([ TestClass(*range(4)),
             TestClass(*range(1,5)),
             TestClass(*range(3,8)),
             TestClass(*range(6,11))
           ])
print(objects[0])
```

now we can actually dive in and access sub-attibutes of our class

```
objects.find_all_where(contains="a") # zero level search (just a modifier)

objects.find_all_where(a=3) # single level search
objects.find_all_where(a__in=[3,6]) # single level search with modifier

objects.find_all_where(a=3) # single level search
objects.find_all_where(a__in=[3,6]) # single level search with modifier

objects.find_all_where(b_list__b=3) # 2nd level search
objects.find_all_where(b_list__b__not_in=[3,5]) # 2nd level search with negated␣
↪modifier

objects.find_all_where(b_list__c__val=4) # 3rd level search
objects.find_all_where(b_list__c__val__gt=7) # 3rd level search with negated modifier
```

you can continue indefinately ... although i imagine the deeper you have to go the slower it will be, but it should be fine for smallish lists

See also:

Query Reference

Lookups Than Span Sub-Objects

`SearchableCollection.find_all_where()`

- genindex
- search

3.3 SearchableCollection API Documentation

In General SearchableCollection attempts to mimic the functionality of a list exactly

that means you can do indexing like `my_list[0], my_list[-1]`

and you can also do slicing like `my_list[5:15:3]`

and you can do standard list setitems like `my_list[6] = SomeClass()`

you can also use the normal `x in my_list` operator

See also:

QUERY ARGUMENTS

Lookups CheatSheet

3.3.1 Available Methods

classmethod `SearchableCollection.`**`find_one_where`**`(`***query_conditions*`)`

> **Parameters** **`query_conditions`** (SEE: *QUERY ARGUMENTS*) – keyword pairs that describe the current search criteria
>
> **Returns**
>
> A single match from the collection (the *first* match found), *or None if no match is found*
>
> search the collection and return the first item that matches our search criteria

```
my_collection.find_one_where(sn="123123",in_use=False)
```

classmethod `SearchableCollection.`**`find_all_where`**`(`***query_conditions*`)`

> **Parameters** **`query_conditions`** (SEE: *QUERY ARGUMENTS*) – keyword pairs that describe the current search criteria
>
> **Returns** all of the matches from the collection
>
> **Return type** generator

this will search the collection for any items matching the provided criteria

```
for result in my_collection.find_all_where(condition1=3, condition2=4):
    do_something(result)
```

classmethod `SearchableCollection.`**`delete_where`**`(`***query_conditions*`)`

> **Parameters** **`query_conditions`** (SEE: *QUERY ARGUMENTS*) – keyword pairs that describe the current search criteria
>
> **Returns** None

Deletes any items in the collection that match the given search criteria

```
my_collection.delete_where(sn__startswith="AB") # delete all things that have a␣
↪sn attribute starting with "AB"
```

- genindex
- search

3.4 Query Reference

Field lookups are how you specify the meat of a query. They're specified as keyword arguments to the following SearchableCollection methods

See also:

Method `find_all_where(**query_conditions)` Documentation of the `SearchableCollection.find_all_where()` method

Method `find_one_where(**query_conditions)` Documentation of the `SearchableCollection.find_one_where()` method

Method `delete_where(**query_conditions)` Documentation of the `SearchableCollection.delete_where()` method

Basic lookups **conditions arguments take the form **<field>__<lookuptype>=value**. (That's a double-underscore).

For example:

```
>>> entry_objects.filter(pub_date__lte=datetime.now())
```

would find all the *things* in entry_objects where `entry_object.pub_date <= now()`*

**if `entry_object` is a dict it would find all entries where `entry_object['pub_date'] <= now()`

additionally you can **negate** any of the lookuptypes by prepending `not_`

```
>>> entry_objects.filter(pub_date__not_lte=datetime.now())
```

would find all entry_objects where `entry_object.pub_date` **IS NOT** less than or equal to `now()`

- *you* ***do not** have to supply both the field and the lookuptype*
- *if you ommit the* **lookuptype***, it will default to* `eq`
- *if you ommit the* **field**, *it will default to the root level object*
- *if you ommit either, you do not need the double underscore(__)*

See also:

Lookups CheatSheet

3.4.1 Query LookupType Reference

eq

tests a field for equality, this is the default lookuptype if None is specified

```
>>> entry_objects.find_all_where(serial_number__eq="SN123123")
>>> entry_objects.find_all_where(serial_number="SN123123")
```

are both equivelent statements, however when using the negated form you *must* specify `eq`

```
>>> entry_objects.find_all_where(serial_number__not_eq="SN123123")
```

is the negated form.

String LookupTypes

`contains`

tests a field to see if it contains a value (or substring)

```
>>> author_objects.find_all_where(articles_id_list__contains=15)
```

would return all the `author_objects`, that had field named `articles_id_list`, that contained the article_id of 15

```
>>> author_objects.find_all_where(articles_id_list__not_contains=15)
```

would return all the `author_objects`, that had field named `articles_id_list`, that DID NOT contain the article_id of 15

`in`

tests a field for membership in a set.

```
>>> entry_objects.find_all_where(status__in=["PENDING","ACTIVE"])
>>> entry_objects.find_all_where(status__not_in=["CANCELLED","FAILED"])
```

note: *if you ommit the* **field** *you must access this as* `is_in`

```
>>> entry_objects.find_all_where(is_in=[1,3,7,9])
```

`startswith`

tests a field for startswith

```
>>> entry_objects.find_all_where(serial_number__startswith("SN76"))
```

finds all the objects with a `serial_number` attribute that starts with `"SN79"`

```
>>> entry_objects.find_all_where(serial_number__not_startswith("SN76"))
```

finds all the objects that **DO NOT** have a `serial_number` attribute that starts with `"SN79"`

`endswith`

tests a field for endswith

```
>>> entry_objects.find_all_where(serial_number__endswith("3"))
```

finds all the objects with a `serial_number` attribute that ends with `"3"`

```
>>> entry_objects.find_all_where(serial_number__not_endswith("3"))
```

finds all the objects that **DO NOT** have a `serial_number` attribute that ends with `"3"`

`search`

tests a field for re.search, that is searches can appear anywhere in the target

```
>>> entry_objects.find_all_where(serial_number__search("3[0-9]"))
```

finds all the objects with a `serial_number` attribute that contains 3 followed by any digit

```
>>> entry_objects.find_all_where(serial_number__not_search("3[0-9]"))
```

finds all the objects that **DO NOT** have a `serial_number` attribute that contains 3 followed by any digit

match

tests a field for re.match, that is matches only match from the beginning

```
>>> entry_objects.find_all_where(serial_number__match("3[0-9]"))
```

finds all the objects with a `serial_number` attribute that starts with a 3 followed by any digit

```
>>> entry_objects.find_all_where(serial_number__not_match("3[0-9]"))
```

finds all the objects that **DO NOT** have a `serial_number` attribute that starts with a 3 followed by any digit

General LookupTypes

lt

less than

```
>>> entry_objects.find_all_where(cost__lt(3.50)) # x < 3.50
>>> entry_objects.find_all_where(cost__not_lt(3.50)) # x >= 3.50
```

lte

less than or equal

```
>>> entry_objects.find_all_where(cost__lte(3.50)) # x <= 3.50
>>> entry_objects.find_all_where(cost__not_lte(3.50)) # x > 3.50
```

gt

greater than

```
>>> entry_objects.find_all_where(rating__gt(9)) # x > 9
>>> entry_objects.find_all_where(rating__not_gt(9)) # x <= 9
```

gte

greater than or equal

```
>>> entry_objects.find_all_where(rating__gte(9)) # x >= 9
>>> entry_objects.find_all_where(cost__not_gte(9)) # x < 9
```

3.4.2 Lookups Than Span Sub-Objects

SearchableCollections offer a powerful and intuitive way to “follow” relationships in lookups, taking care of the search for you automatically, behind the scenes. To span a sub-object, just use the field name of sub-objects, separated by double underscores, until you get to the field you want.

```
>>> entry_objects.filter(blog__name='Beatles Blog')
```

this assumes you have an object with a field named “blog”, blog has a field named “name”

```
>>> entry = {"blog":{"name":...,"date":...,"author":{"name":...,"publications":[...]}}
```

this will locate the entry that has a blog, with a name field of “Beatles Blog”

This spanning can be as deep as you like

```
>>> entry_objects.filter(blog__author__name='Lennon')
```

- genindex
- search

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index